Dealing with Big Data: Streaming, HOF, and Parallelism

CUNY The City College of New York
NYU Center for Urban Science + Progress

Streaming Summary

- Benefits of Streaming:
  - Reduce the memory footprint
  - Enable pipelining, aka. pushing data out as soon as possible
- Challenges of Streaming:
  - Not every algorithm is streamable
  - Many are not trivial to approximate
- Handling data streams in Python:
  - Generators and Iterators

generator

- ***generators*** are special classes for creating *iterators* that can only be traversed once.

```
A = (i for i in xrange(5))
print 'OUTPUT:',
for i in A:
    print i,
print '\nOUTPUT:',
for i in A:
    print i,
# OUTPUT: 0 1 2 3 4
# OUTPUT:
```

- generators can be used as data streams as it usually **doesn't store** everything in memory

How to create a generator?

- generators are functions BUT use the keyword **yield** in place of **return**
- Example: create a generator of square numbers from 1 to 100
- The code in function body only runs each time the for uses the generator to "generate" data
- The generator is done when the function finishes

```
def square_numbers():
    for i in xrange(10):
        yield i**2

numbers = square_numbers()
print 'FIRST:',
for i in numbers:
    print i,
print '\nSECOND:',
for i in numbers:
    print i,
# FIRST: 0 1 4 9 16 25 36 49 64 81
# SECOND:
```

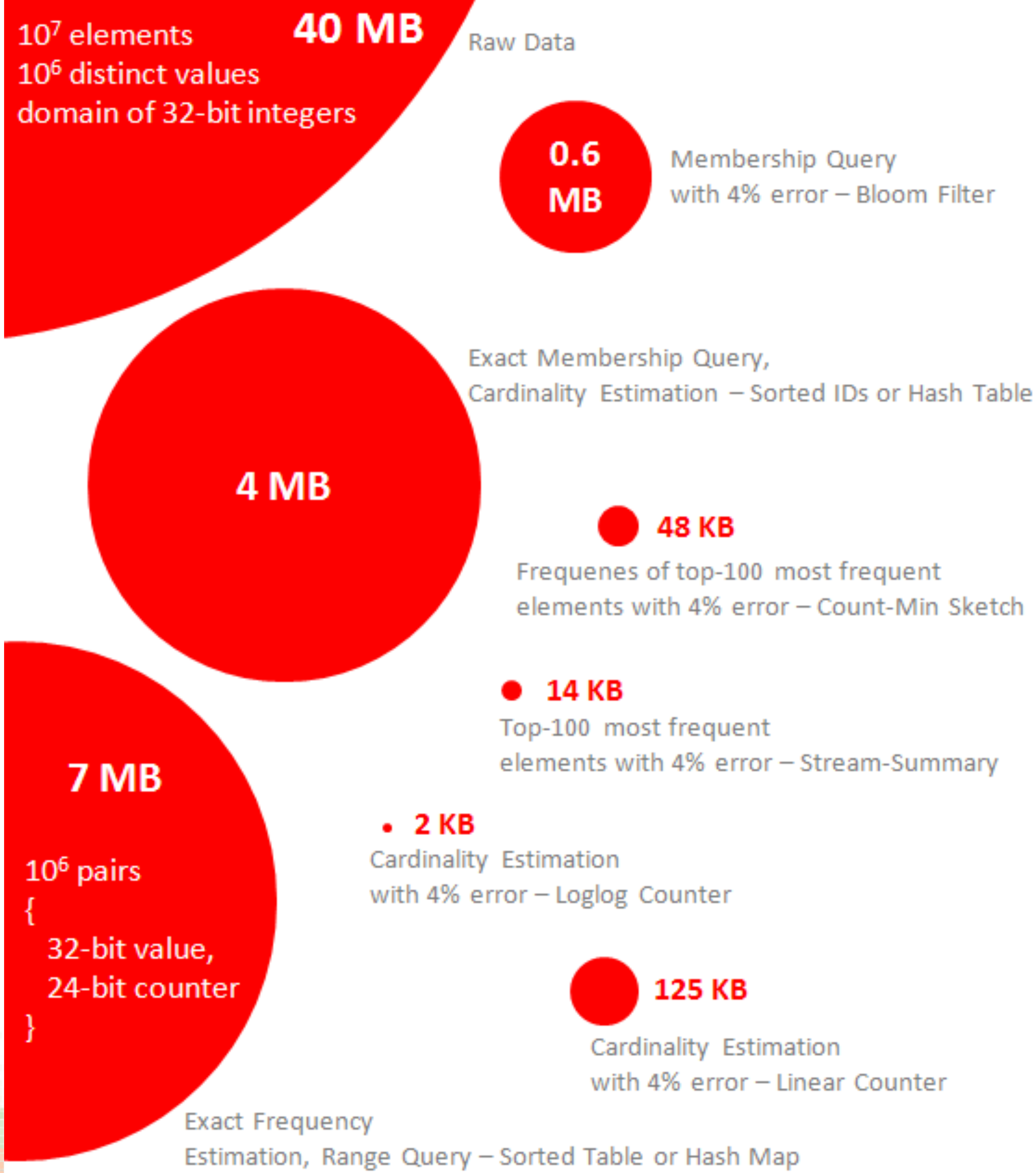
Good read on Sketching

- *Probabilistic Data Structures for Web Analytics and Data Mining*, Ilya Katsov, 2012.

  https://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/

- *A practical introduction to the Count-Min Sketch*, Hannes Korte, 2013.

  http://hkorte.github.io/slides/cmsketch/



[Katsov, I. 2012]

Linear Counting

```
class LinearCounter {
    BitSet mask = new BitSet(m) // m is a design parameter

    void add(value) {
        int position = hash(value) // map the value to the range 0..m
        mask.set(position) // sets a bit in the mask to 1
    }
}
```

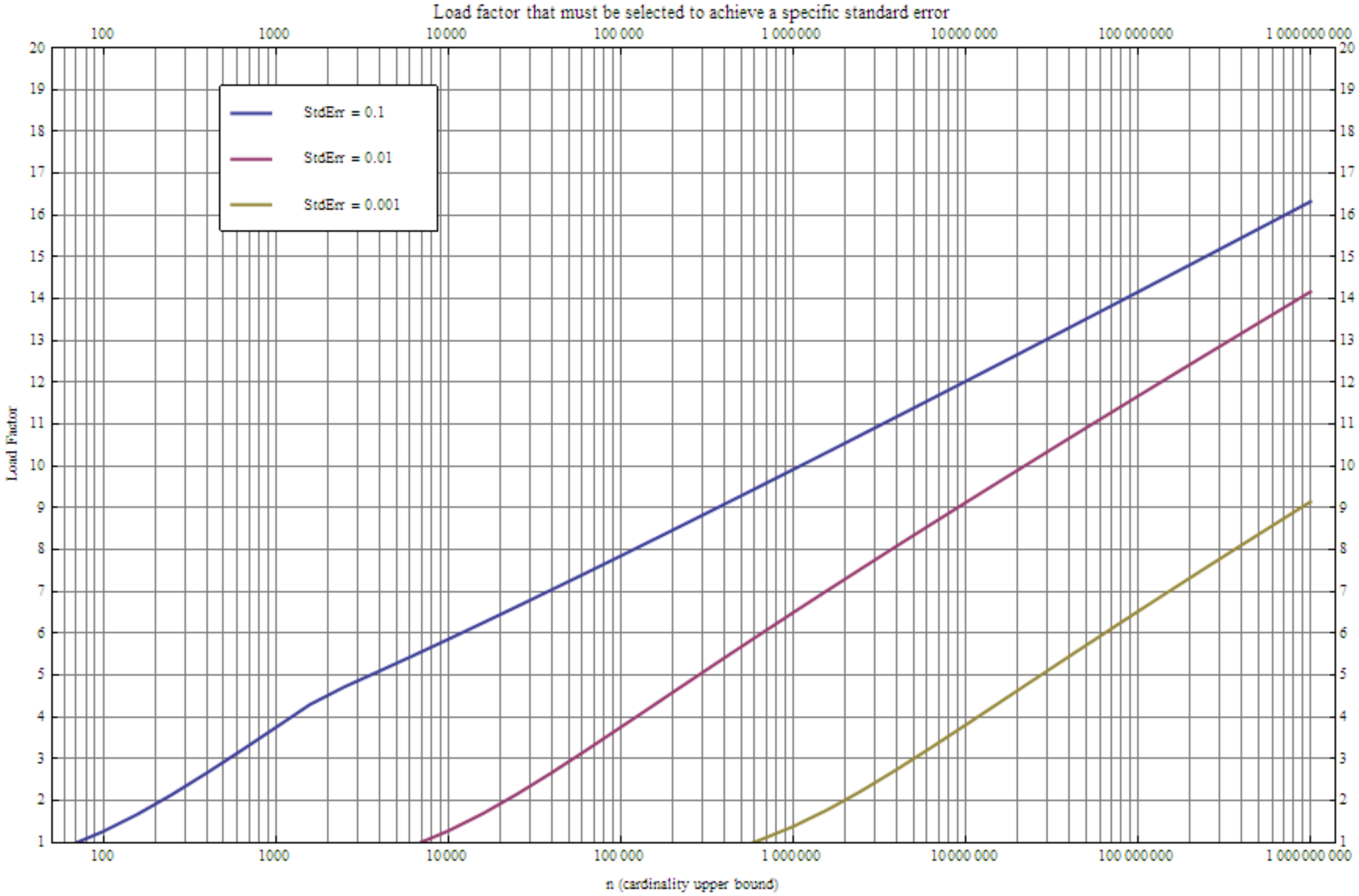
[Katsov, I. 2012]

Linear Counting

$\hat{n} = -m \ln \frac{m-w}{m}$	$\hat{n}$ - cardinality estimation w - mask weight (a number of 1's) m - mask size
$bias(\frac{\hat{n}}{n}) = E(\frac{\hat{n}}{n}) - 1 = \frac{e^t - t - 1}{2n}$	This equation expresses a bias of the estimation (the ratio between estimation and true cardinality) as a function of the load factor and expected cardinality (or upper bound). t - load factor, n/m E(.) - mathematical expectation n - maximum cardinality (or upper bound, or capacity)
$m > \max(5, 1/(\varepsilon t)^2) \cdot (e^t - t - 1)$	A practical formula that allow one to choose m by the standard error of the estimation. m - mask size ε - standard error of the estimation t - load factor, n/m

[Katsov, I. 2012]

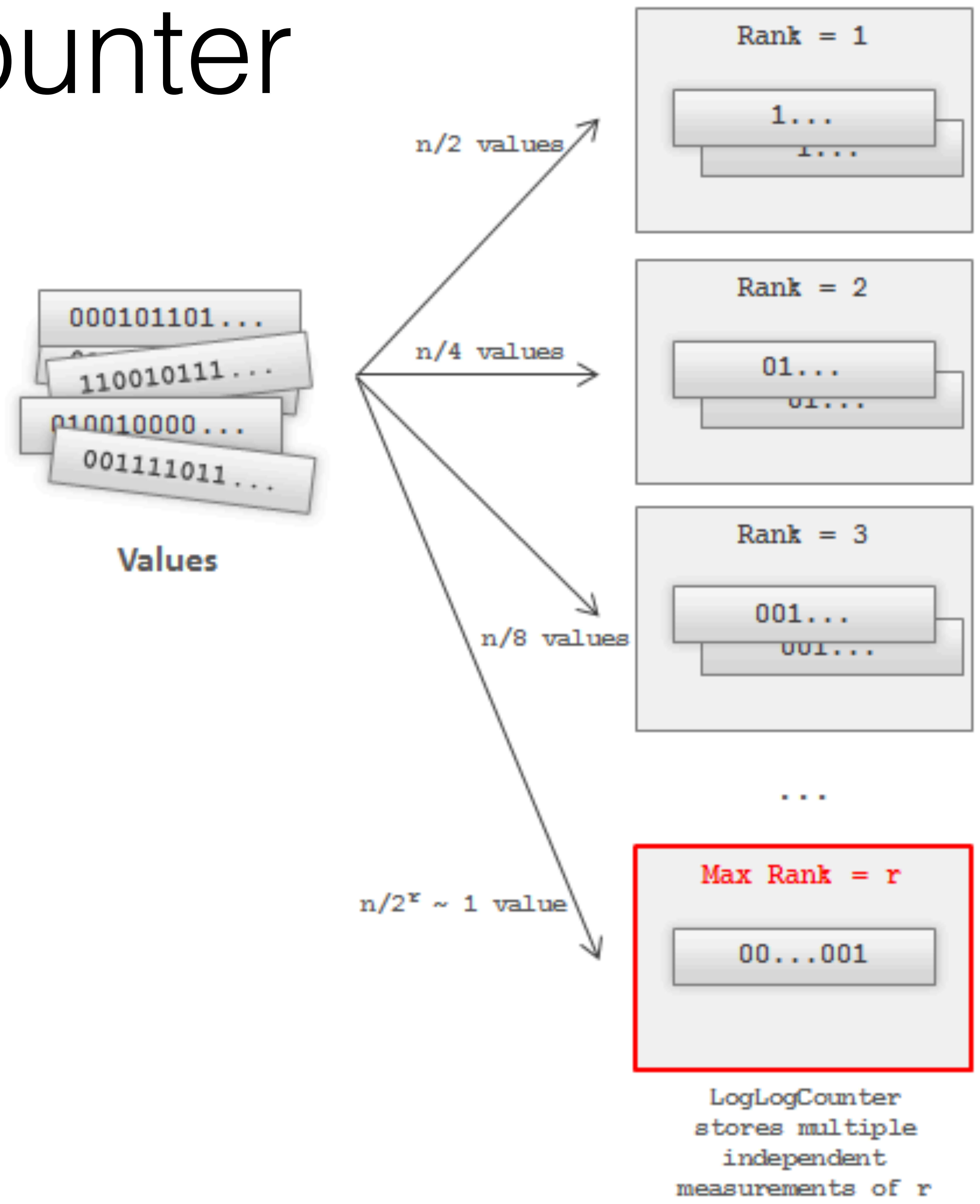
Linear Counting



[Katsov, I. 2012]

LogLog Counter

- Tracks the rarest element in the data set
- Estimate the cardinality based on this information (the rank — **r**)
- Intuition: getting all heads in flipping coins, what are the odds?
- The rank value **r** has high variance
  - Count multiple of them and take the mean value.



[Katsov, I. 2012]

LogLog Counter

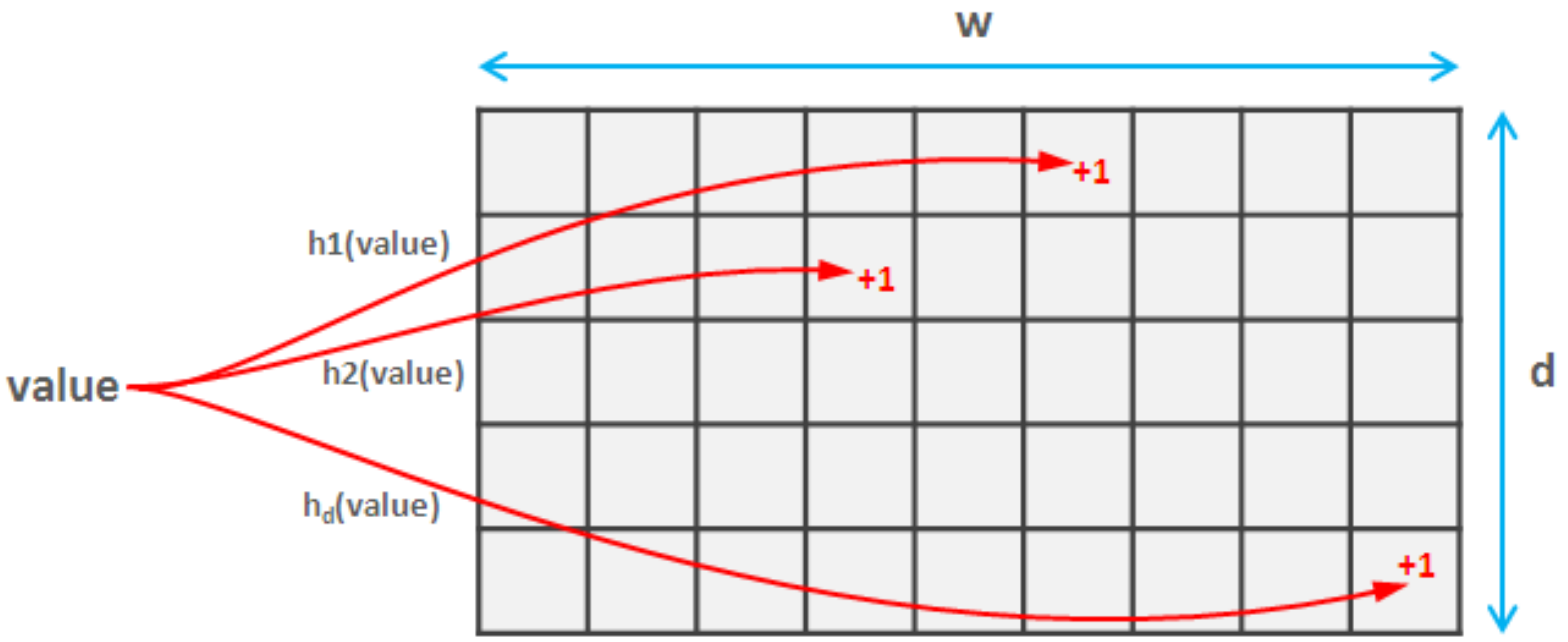
- 1024 estimators ~ 4% std error
- H is the hashed length (in bits)
- 5-bit bucket (m=32) can support sets of cardinalities of 10 billions.
- 8-bit bucket (m=256) can support extremely large cardinalities.
- HyperLogLog = LogLog + Harmonic Mean (lower std error)

$\hat{n} = \alpha_m \cdot m \cdot 2^{1/m \sum_j estimators[j]}$ $\alpha_m = \Gamma(\frac{-1}{m})\frac{1-2^{\frac{1}{m}}}{\ln 2} \overset{m>64}{\approx} 0.39701$	$\hat{n}$ – cardinality estimation m – number of buckets (estimators) α_m – estimation factor, close to 0.39701 for m > 64, i.e. for most of practical applications
$\varepsilon \approx \frac{1.30}{\sqrt{m}}$	Dependency between the standard error of the estimation and the number of buckets (estimators). ε - standard error of the estimation m – number of buckets (estimators)
$H = \log_2 m + \lceil \log_2(n/m) + 3 \rceil$	A practical formula for length of the hash function. m – number of buckets (estimators) n – maximum cardinality (i.e. capacity)
$etype \Leftarrow \lceil \log_2 \lceil \log_2(n/m) + 3 \rceil \rceil$	A number of bits in etype is determined by the maximal possible rank. The rank is limited by H, so the length of etype is a log of H (except the part that is used for bucket ID computation).

[Katsov, I. 2012]

Count-Min Sketch

```
class CountMinSketch {
    long estimators[][] = new long[d][w]    // d and w are design parameters
    long a[] = new long[d]
    long b[] = new long[d]
    long p      // hashing parameter, a prime number. For example 2^31-1

    void initializeHashes() {
        for(i = 0; i < d; i++) {
            a[i] = random(p)    // random in range 1..p-1
            b[i] = random(p)
        }
    }

    void add(value) {
        for(i = 0; i < d; i++)
            estimators[i][ hash(value, i) ]++
    }

    long estimateFrequency(value) {
        long minimum = MAX_VALUE
        for(i = 0; i < d; i++)
            minimum = min(
                minimum,
                estimators[i][ hash(value, i) ]
            )
        return minimum
    }

    hash(value, i) {
        return ((a[i] * value + b[i]) mod p) mod w
    }
}
```

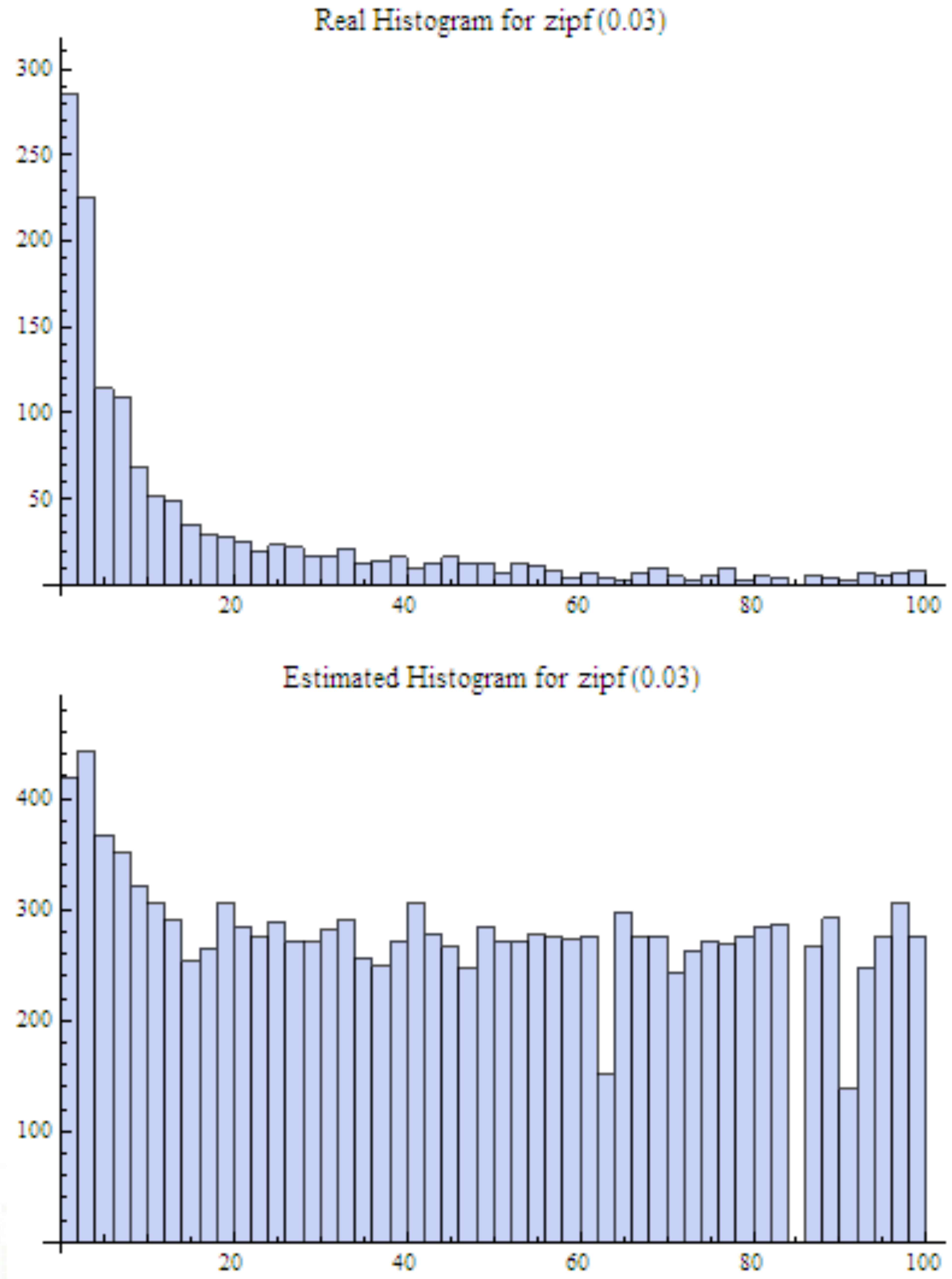


estimation error $\varepsilon \le 2n/w$ *with probability* $\delta = 1 - (1/2)^d$	n – total count of registered events w – sketch width d – sketch height (aka depth)

[Katsov, I. 2012]

Count-Min Sketch

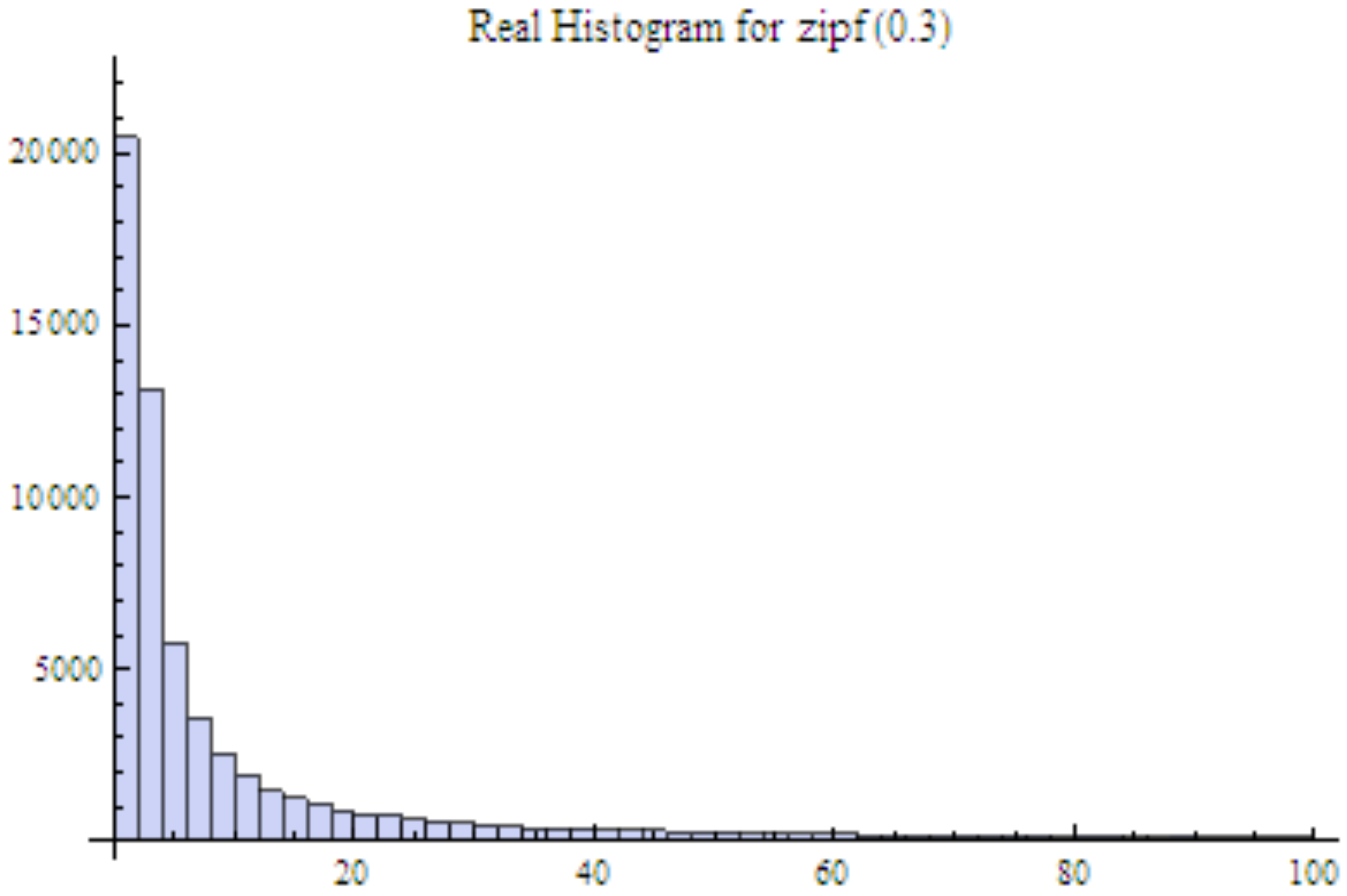
- 192 counters
- 10,000 elements
- 8,500 distinct values
- Find 100 most frequently used words
- Not so good estimation

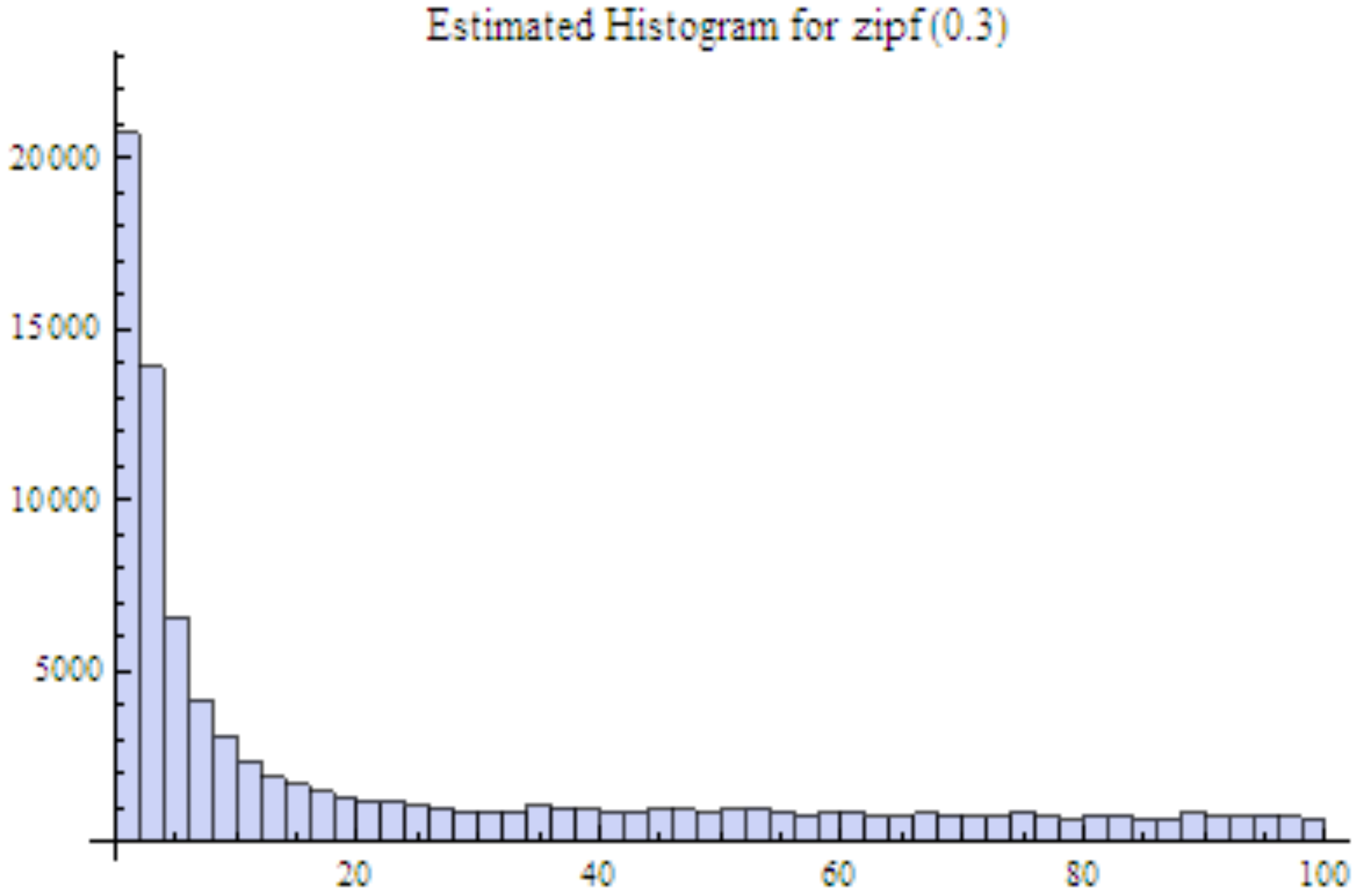


[Katsov, I. 2012]

Count-Min Sketch

- 192 counters
- 80,000 elements
- 8,500 distinct values
- Find 100 most frequently used words
- Much better for highly skewed data



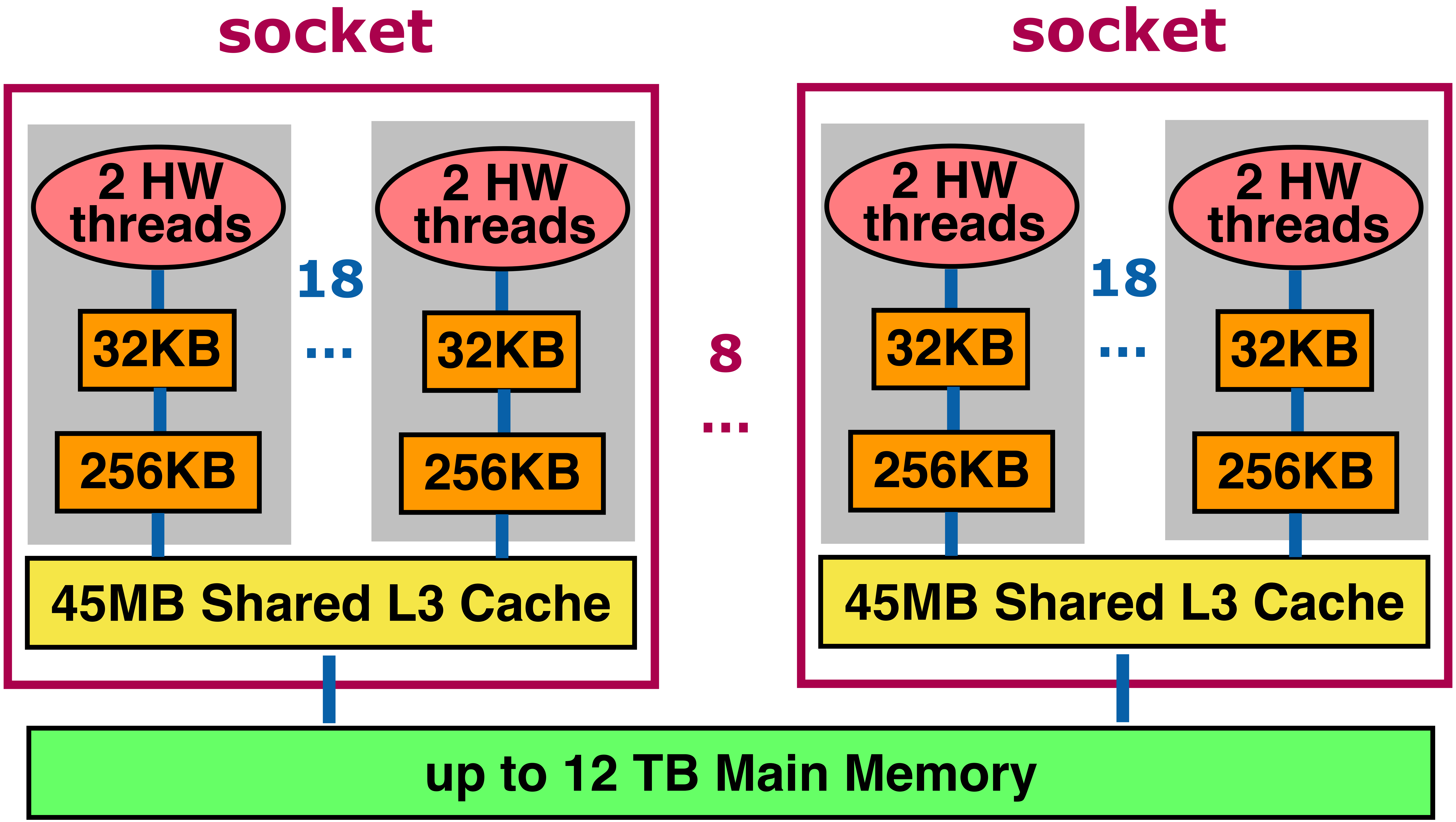


[Katsov, I. 2012]

Big Data Computing Model: Parallelism

- Scale up is much faster than Scale out
  - Minimal network communication (slowest in the bandwidth hierarchy)
  - Can fit even several TBs of RAM (compute@cusp has 1TB!)
  - Fast access for spatial (pre-fetch) and temporal (caching) locality
- Leverage multi-core and many-core architectures
  - Need to exhaust each machine resources before going across nodes
  - Hierarchical (non-uniform) memory access matters!

Multicore: 144-core Xeon Haswell E7-v3



Attach: Hard Drives & Flash Devices

Common Big Data Challenges

- Volume:
  - ***Too much data to process*** — reduce processing time by using distributed & parallel computing (next class)
    - Performing OCR on 1000s of articles simultaneously
  - ***Too big to fit in RAM*** (esp. when no cluster resource available)
    - Given 30GB of taxi trip records → *how to plot the trend?*
- Velocity — ***data comes in real-time*** → too much to store → process **on the fly**!
  - Detecting network failures from inspecting billions of packets per hour

Common Big Data Challenges

- Volume:
  - ***Too much data to process*** — reduce processing time by using distributed & parallel computing (next class)
    - Performing OCR on 1000s of articles simultaneously
  - ***Too big to fit in RAM*** (esp. when no cluster resource available)
    - Given 30GB of taxi trip records → *how to plot the trend?*

Streaming Computation

- Velocity - s **on the fly**!
  - Detecting network failures from inspecting billions of packets per hour

Common Big Data Challenges

- Volume:
  - ***Too much*** distributed & parallel cor

Parallel Computing

  - Performing OCR on 1000s of articles simultaneously
  - ***Too big to fit in RAM*** (esp. when no cluster resource available)
    - Given 30GB of taxi trip records → *how to plot the trend?*

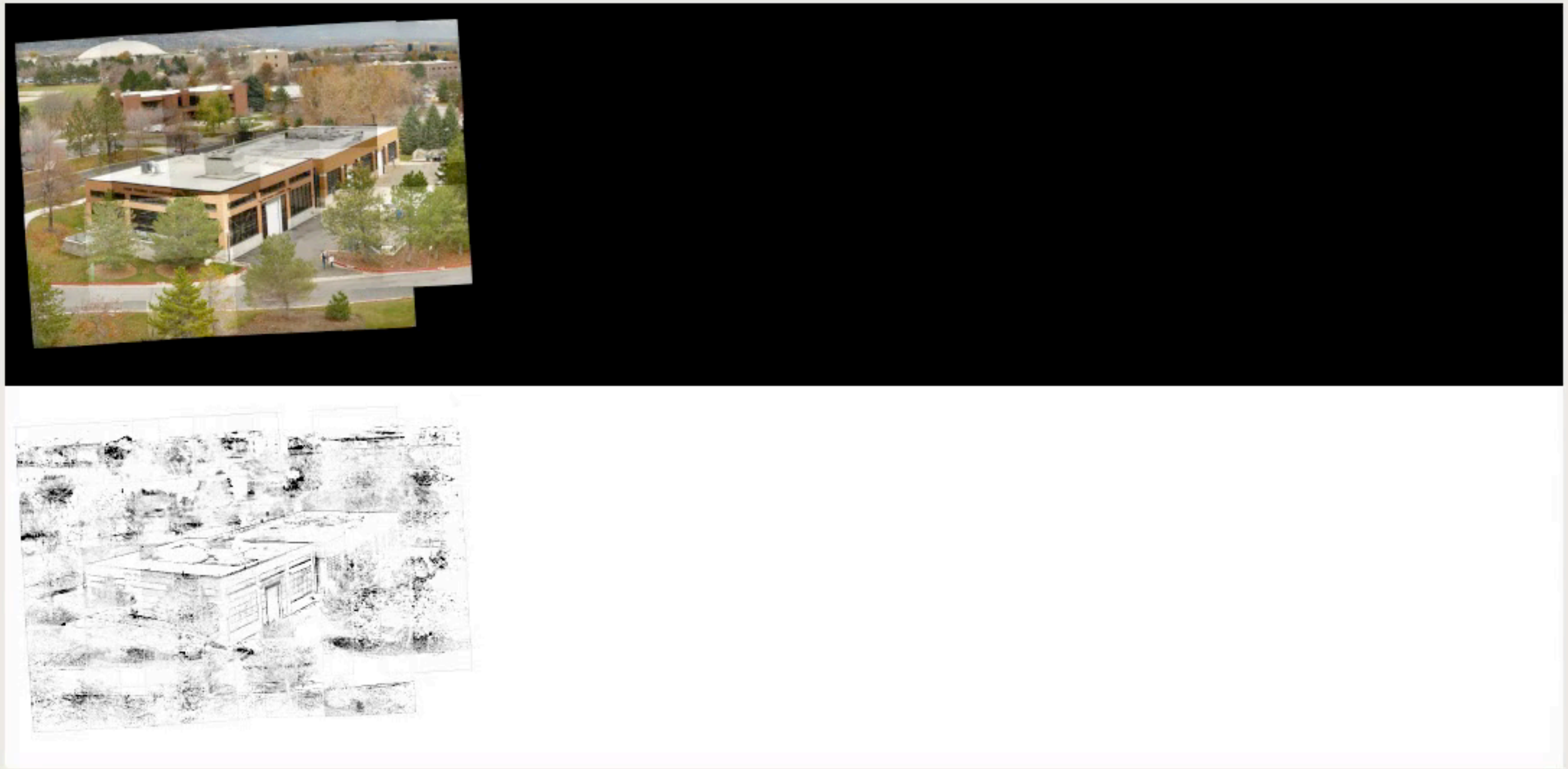
Streaming Computation

- Velocity - s **on the fly**!
  - Detecting network failures from inspecting billions of packets per hour

Parallel Computing

- Make use of parallel architectures (e.g. multi-core, multi-processor, clusters of machines, etc.) to improve computing performance
  - Using multiple cores to speedup video processing (Facetime HD!)
  - Using multiple processors in web servers
  - Using GPUs to increase rendering frame rate in games
  - Using clusters to run simulations
- Have a long history in High Performance Computing (what is the difference?)
  - Similar to streaming, getting popular in the *Big Data* era

Panoramic Edge Detection



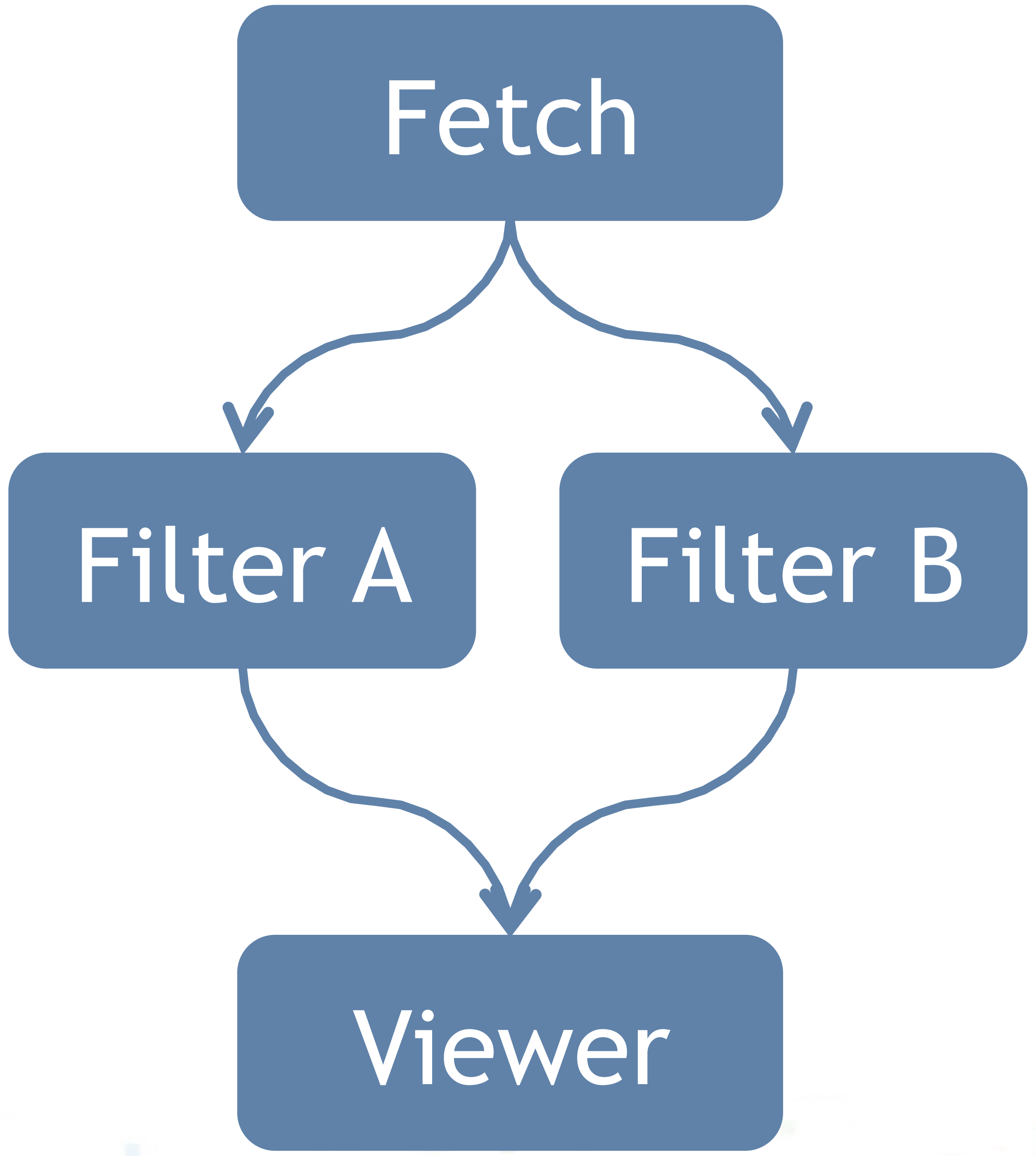
HyperFlow Execution Engine Status

Virtual Processing Element	Executing
Tesla C1060	Gaussian Blur
GeForce GTX 295	Auto Levels
GeForce GTX 295	idle
CPU Core 0 Thread 0	Decode Image
CPU Core 1 Thread 0	Pixel Threshold
CPU Core 2 Thread 0	Decode Image
CPU Core 3 Thread 0	Pixel Threshold
CPU Core 0 Thread 1	Decode Image
CPU Core 1 Thread 1	idle
CPU Core 2 Thread 1	Pixel Threshold
CPU Core 3 Thread 1	Decode Image

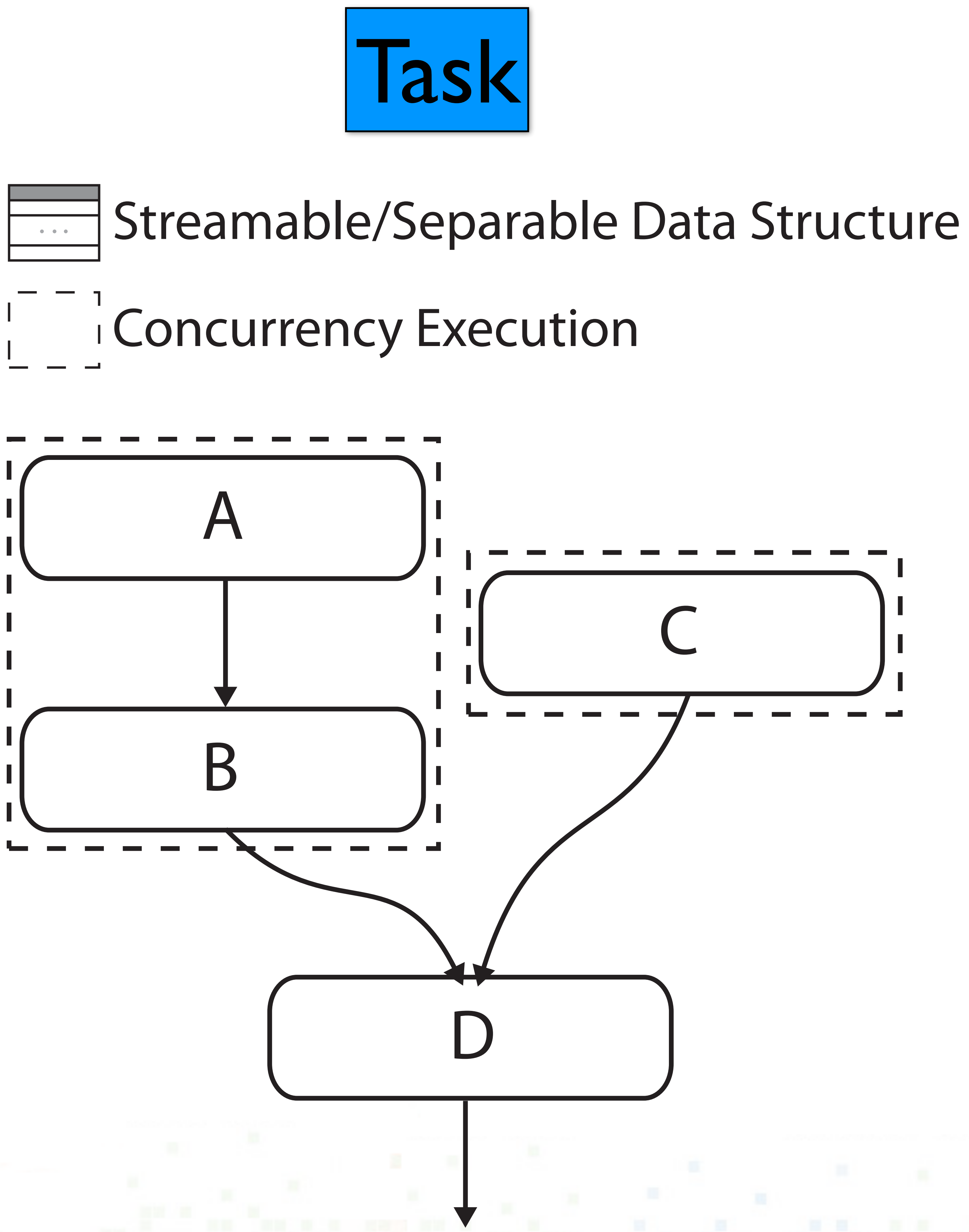

Panoramic Edge Detection
HyperFlow Execution Engine Status
Virtual Processing Element
Executing
Tesla C1060
Gaussian Blur
GeForce GTX 295
Auto Levels
GeForce GTX 295
idle
CPU Core 0 Thread 0
Decode Image
CPU Core 1 Thread 0
Pixel Threshold
CPU Core 2 Thread 0
Decode Image
CPU Core 3 Thread 0
Pixel Threshold
CPU Core 0 Thread 1
Decode Image
CPU Core 1 Thread 1
idle
CPU Core 2 Thread 1
Pixel Threshold
CPU Core 3 Thread 1
Decode Image

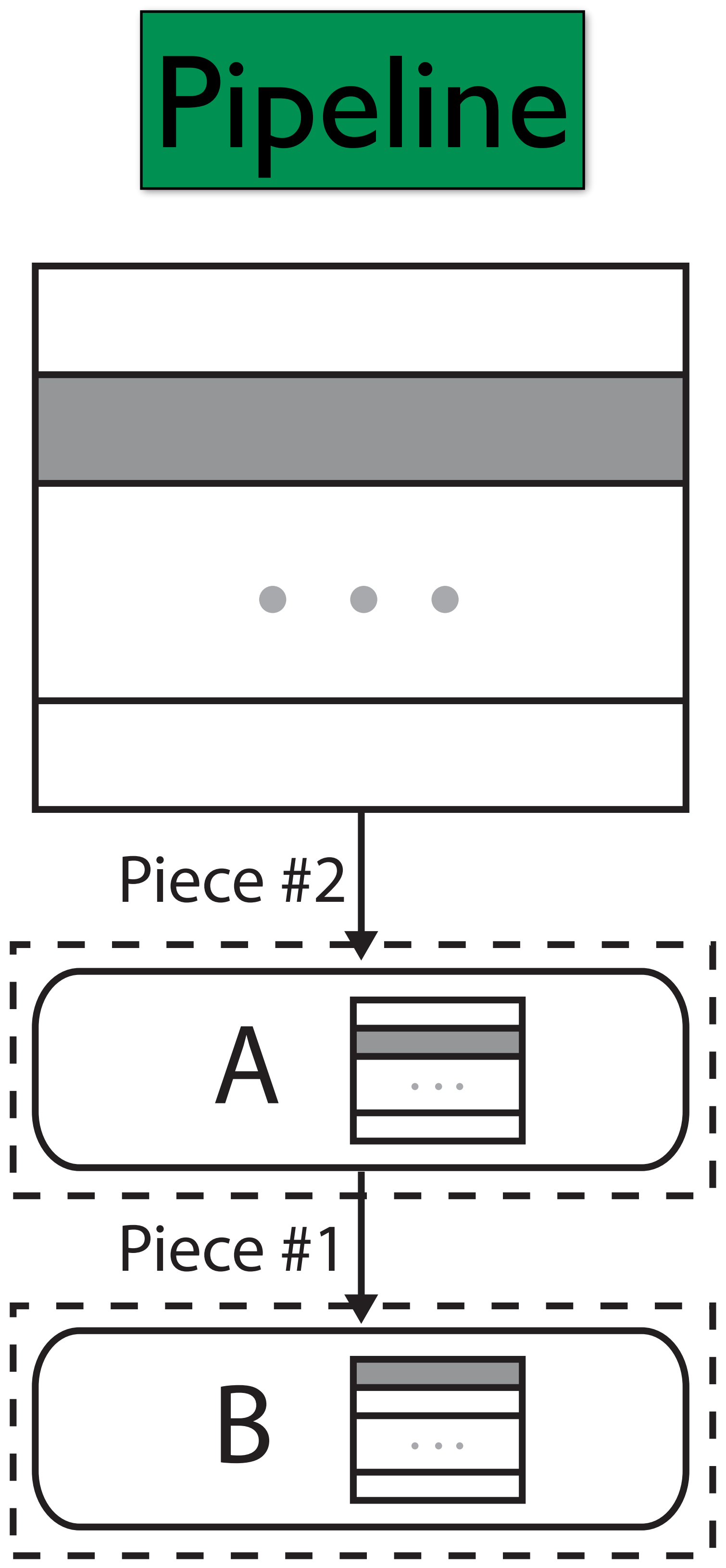
Parallel Computing Model

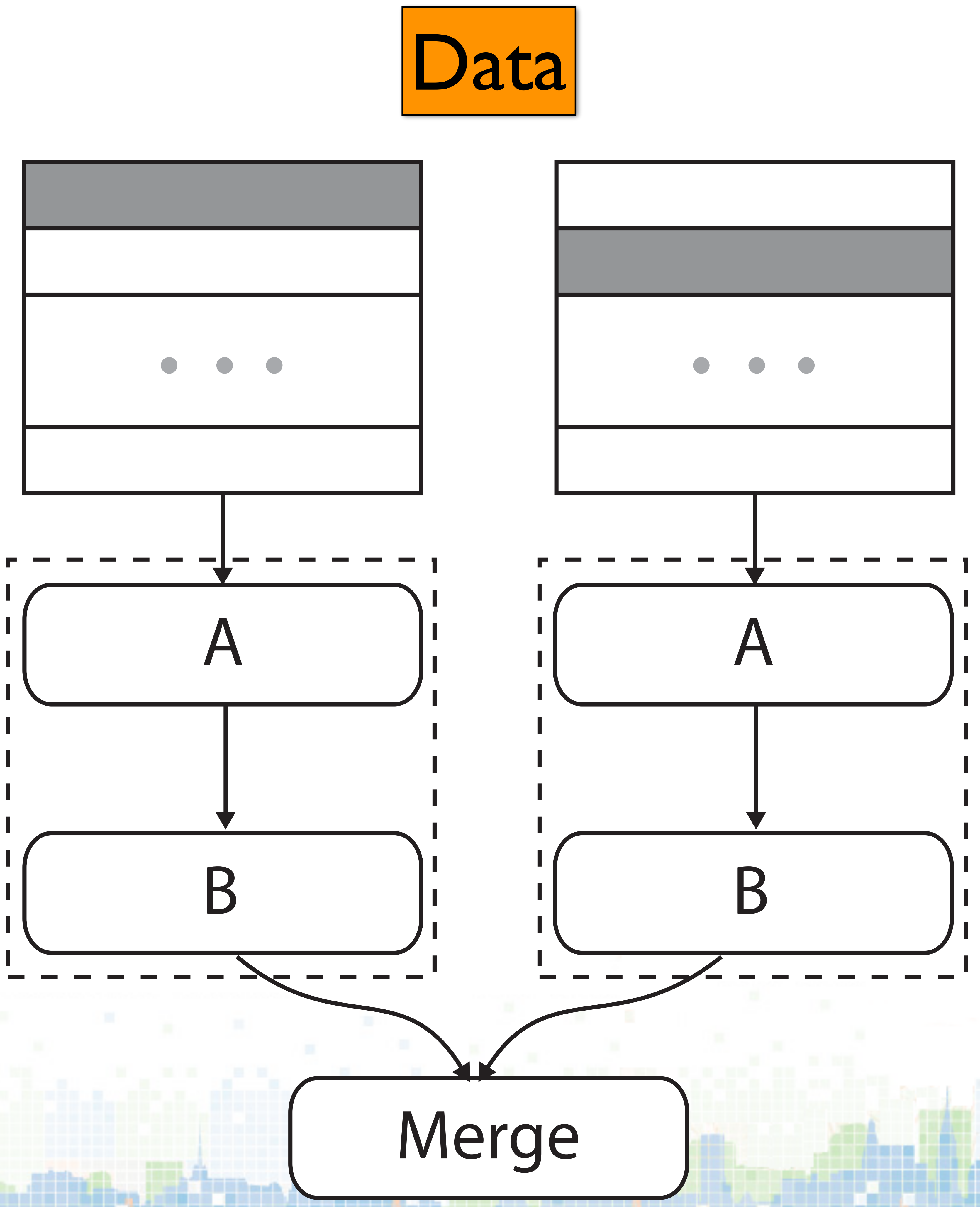
- Tasks are splittable into independent subtasks: either smaller tasks OR tasks that would operate on smaller “chunks” of data
- Data dependency between (sub-)tasks
- Can be specified using pipelines or DAGs
  - Task Graph / Workflow / Dataflow
  - Modules = Computation
  - Connection = Data dependency
- Data dependency ensures execution order



Types of Parallelism in Task Graphs







Task Parallelism

- Distributing tasks to run simultaneously on the same or different data — achieve efficiency by the number of tasks
- Processes communicate through tokens passed in a workflow
- Example:
  - Performing data cleaning on two separate data sets
  - Running linear regression & random forest on two sets of variables

Pipeline Parallelism

- Explicitly allocating resources for each phase of the processing pipeline — achieve efficiency by the number of phases
- Processes communicate through data passed in the pipeline.
- Pipeline ~ Task + Streaming
- Example:
  - Dedicate one process each for data acquisition, data compression, and data encryption in a data ingestion pipeline.

Data Parallelism

- Distributing data to different processors or nodes to run simultaneously — achieve efficiency by the number of nodes
- Processes usually communicate minimally at the end to merge data, thus, considered “embarrassingly Parallel” in many cases
- Examples:
  - Convert birth_year into age, where each record can be processed independently

Pros and Cons of Parallelism

- Task Parallelism
  - Pros : simple to manage, no need for data partitioning
  - Cons : limited by the number of tasks, no data coherency
- Pipeline Parallelism
  - Pros : low memory footprint (sliding window), data coherency
  - Cons : limited by the number of tasks with pipeline bottleneck
- Data Parallelism
  - Pros : scale by the number of data “chunks” — the bigger the better
  - Cons : large memory footprint (each technician must have all resources)

in practice: these are combined + concurrent & distributed computing

Pros and Cons of Parallelism

- Task Parallelism
  - Pros : simple to manage, no need for data partitioning
  - Cons : limited by the number of tasks, no data coherency
- Pipeline Parallelism
  - Pros : low memory footprint (sliding window), data coherency
  - Cons : limited by the number of tasks with pipeline bottleneck
- Data Parallelism
  - Pros : scale by the number of data “chunks” — the bigger the better
  - Cons : large memory footprint (each technician must have all resources)

Parallel vs. Concurrent vs. Distributed Computing

Parallel vs. Concurrent vs. Distributed Computing

- Parallel Computing: taking full advantage of parallel architecture to speed up computation — *all about performance*
  - Data can be distributed to leverage additional processing power
- Concurrent Computing: enabling processes/tasks to progress without waiting for each other — *all about dependencies*
  - Parallel Computing in task graphs requires concurrency
- Distributed Computing: executing computations on distributed systems properly — *all about uncertainty*
  - Data are distributed by nature (e.g. because of their volume)

Big Data Computing

- Big Data requires distributed computing
  - part of the problem
  - to scale out storage and in-database processing
- Big Data needs parallelism (thus, also concurrent) computing
  - focus on data parallelism for its scalability
- Big Data Platform usually
  - expose parallelism through a custom programming model
  - but handle most distributed computing issues behind the scene

Review: General Data Computing Model

- Given a collection of data records (e.g. a large array or list of records)
- Apply a series of *data transformations* to the collection
- Collect the final transformation

birth_year		age		senior (>60)
[1978, 1980, … , 1950]	→	[38, 36, … , 66]	→	[66]

Example: How to do it in Python?

birth_year		age		senior (>60)
[1978, 1980, … , 1950]	→	[38, 36, … , 66]	→	[66]

Example: How to do it in Python?

birth_year		age		senior (>60)
[1978, 1980, … , 1950]	⟶	[38, 36, … , 66]	⟶	[66]

```
age = []
for y in birth_year:
    age.append(2016-y)

senior = []
for y in age:
    if y>60:
        senior.append(y)
```

Example: How to do it in Python?

birth_year [1978, 1980, … , 1950] → age [38, 36, … , 66] → senior (>60) [66]

```
age = []
for y in birth_year:
    age.append(2016-y)

senior = []
for y in age:
    if y>60:
        senior.append(y)
```

```
birth_year = pd.Series(birth_year)

age = 2016-birth_year

senior = age[age>60]
```

Example: How to do it in Python?

name,birth_year
[(‘A’,1978), … , (‘C’,1950)] ⟶ name,age
[(’A’,38), … , (’C’,66)] ⟶ name,senior
(>60 with name ‘C’)
[(’C’,66)]

Example: How to do it in Python?

name,birth_year
[(‘A’,1978), … , (‘C’,1950)] → name,age
[(’A’,38), … , (’C’,66)] → name,senior
(>60 with name ‘C’)
[(’C’,66)]

```
age = []
for y in birth_year:
    age.append((y[0], 2016-y[1]))

senior = []
for y in age:
    if y[0]=='C' and y[1]>60:
        senior.append(y)
```

Example: How to do it in Python?

name,birth_year
[(‘A’,1978), … , (‘C’,1950)] ⟶ name,age
[(’A’,38), … , (’C’,66)] ⟶ name,senior
(>60 with name ‘C’)
[(’C’,66)]

```
age = []
for y in birth_year:
    age.append((y[0], 2016-y[1]))

senior = []
for y in age:
    if y[0]=='C' and y[1]>60:
        senior.append(y)
```

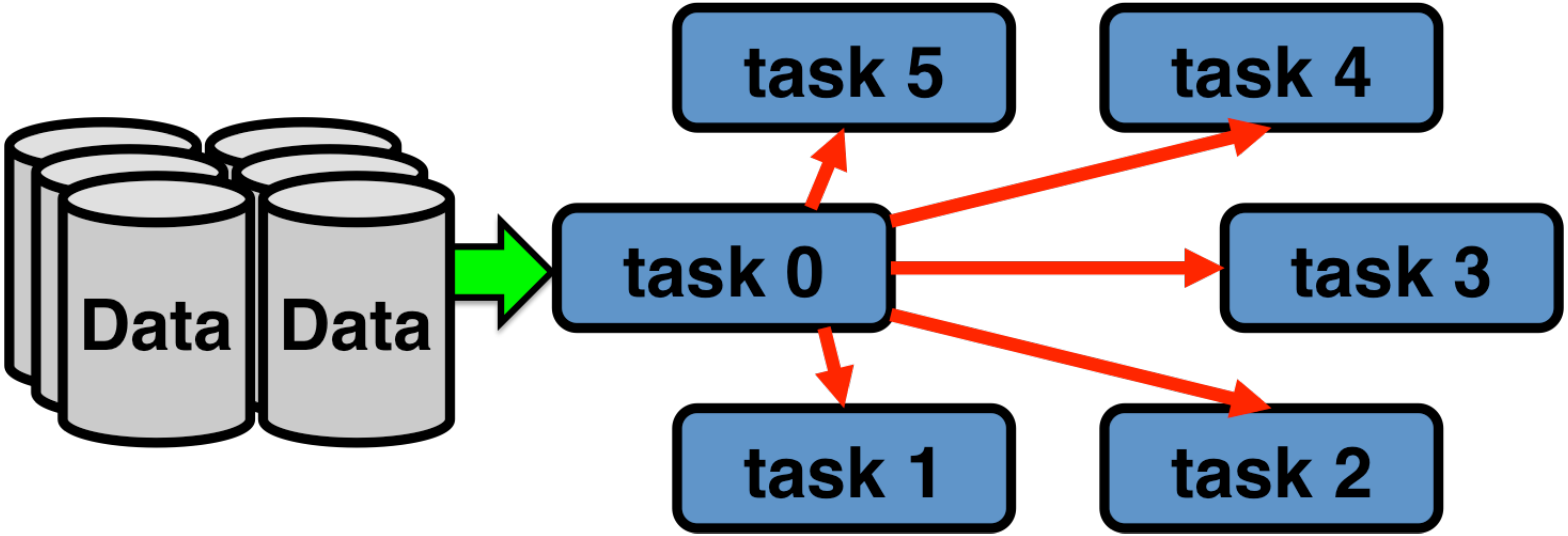
```
birth_year = pd.Series(birth_year)

age = 2016-birth_year

senior = age[age>60]
```

?

The Challenges wrt Big Data

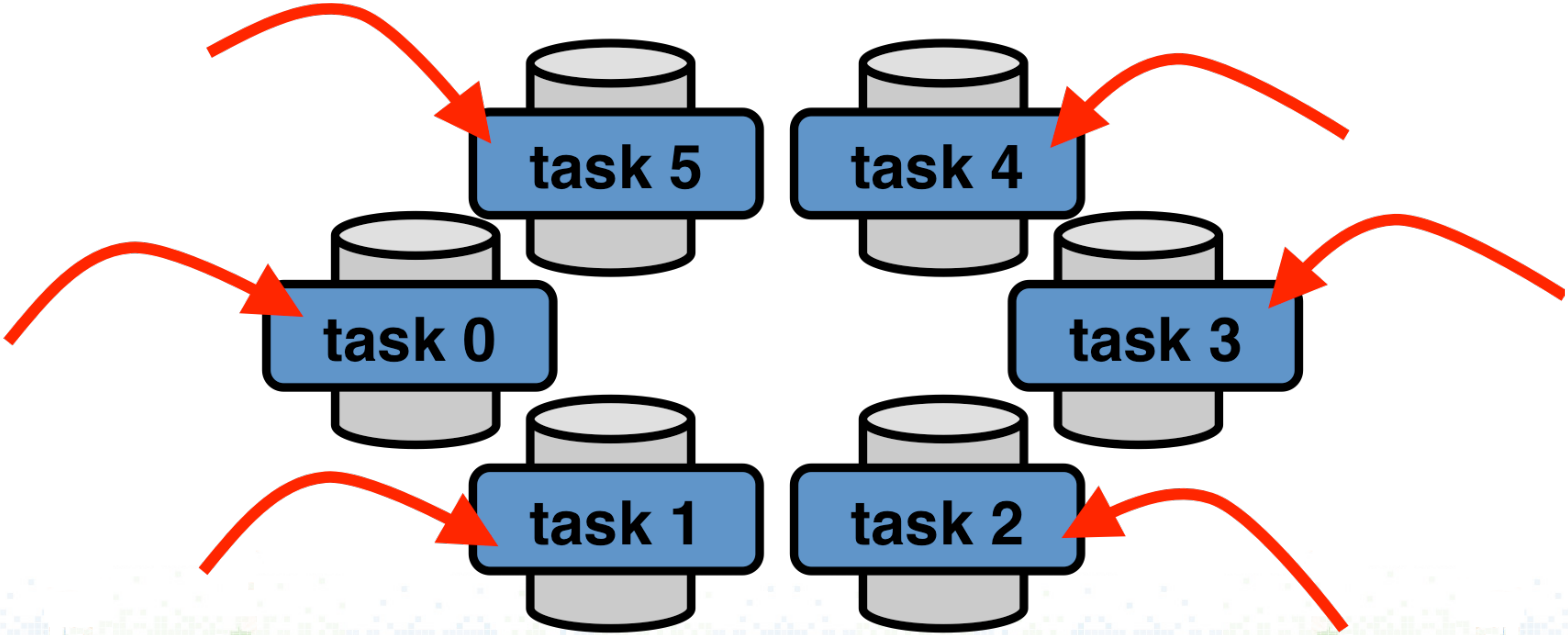
- How to specify these *data transformation* so that the system can:
  - bring compute to the data
  - minimize moving or making changes to data
  - operate on collection of data
  - allow users to dictate *what* to be done to the data and LEAVING *how* it will be done to the platform
  - leveraging both scaling up and scaling out capabilities

Traditional Parallel Computing Model (HPC)



[Glenn K. Lockwood. Conceptual overview of map-reduce and hadoop]

Big Data Computing Model



[Glenn K. Lockwood. Conceptual overview of map-reduce and hadoop]

The Challenges wrt Big Data

- How to specify these *data transformation* so that the system can:
  - bring compute to the data
  - minimize moving or making changes to data
  - operate on collection of data
  - allow users to dictate *what* to be done to the data and LEAVING *how* it will be done to the platform
  - leveraging both scaling up and scaling out capabilities

The Challenges wrt Big Data

- How to specify these *data transformation* so that the system can:
  - bring compute to the data
  - minimize moving or making changes to data
  - operate on collection of data
  - allow users to dictate *what* to be done to the data and LEAVING *how* it will be done to the platform
  - leveraging both scaling up and scaling out capabilities

a declarative and/or functional language for data processing

Higher-Order Functions (Functional)

- Higher-Order functions (functional) are functions that:
  - take functions as arguments
  - **AND/OR** return functions as its results
- HOF is used to abstract common iteration operations
  - *focus on the* **what** *instead of the* **how**
- Using a predefined set of Higher-Order Functions, we can apply data transformation to data using our own **function** (aka transformation)
  - similar to streaming data elements through our function

Example: How to do it i using HOF?

```
age = []
for y in birth_year:
    age.append(2016-y)

senior = []
for y in age:
    if y>60:
        senior.append(y)
```

Example: How to do it i using HOF?

```
def AGE_TRANSFORM(y):
    return 2016-y

def AGE_FILTER(y):
    return y>60


age = []
for y in birth_year:
    age.append(AGE_TRANSFORM(y))

senior = []
for y in age:
    if AGE_FILTER(y):
        senior.append(y)
```

Example: How to do it i using HOF?

```
def AGE_TRANSFORM(y):
    return (y[0], 2016-y[1])

def AGE_FILTER(y):
    return y[0]=='C' and y[1]>60


age = []
for y in birth_year:
    age.append(AGE_TRANSFORM(y))

senior = []
for y in age:
    if AGE_FILTER(y):
        senior.append(y)
```

Example: How to do it i using HOF?

```
def AGE_TRANSFORM(y):
    return (y[0], 2016-y[1])

def AGE_FILTER(y):
    return y[0]=='C' and y[1]>60


age = map(AGE_TRANSFORM, birth_year)


senior = filter(AGE_FILTER, age)
```

Python's Core HOF

- map(): applies a function over an iterable to produce a new iterable

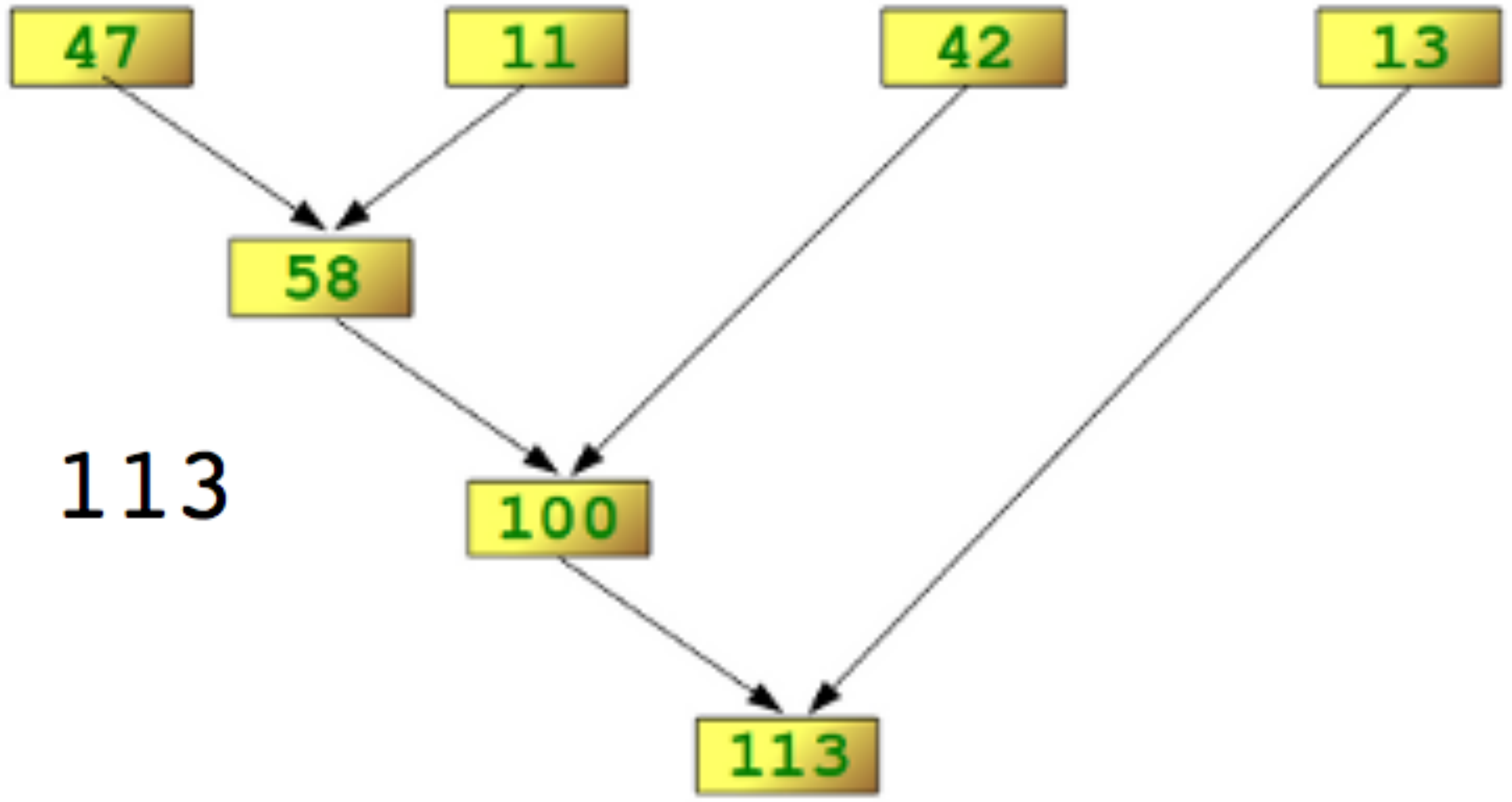
```
map(int, ['0', '1']) -> [0, 1]
```

- filter(): return only values that satisfy a predicate in the new iterable

```
filter(bool, [0, 1]) -> [1]
```

- reduce(): accumulate a sequence of values from left

```
reduce(operator.add, [47,11,42,13]) -> 113
```



- sorted(): return a new sorted list using a comparator
-

Anonymous function: *lambda*

- What if we only use a function once or would like to define it in-place?
- Python's anonymous lambda function:

```
lambda VAR1,VAR2,...,VARN: (expression on VAR1..N)
```

- No name, no return statement, only an expression to evaluate
- Come in handy in defining functions in HOF

```
x2 = lambda x: x*x
x2(10) # 100

x_y = lambda x,y: x+y
x_y(1,2) # 3
```

-

Python's Core HOF

- map(): applies a function over an iterable to produce a new iterable

```
map(lambda x: int(x)+1, ['0', '1']) -> [1, 2]
```

- filter(): return only values that satisfy a predicate in the new iterable

```
filter(lambda x: x<1, [0, 1]) -> [0]
```

- reduce(): accumulate a sequence of values from left to right

```
reduce(lambda x,y: x+y, [47,11,42,13]) -> 113
```

- sorted(): return a new sorted list using a comparator function

```
sorted(xrange(5))                    -> [0,1,2,3,4]
sorted(xrange(5), cmp=lambda x,y:y-x) -> [4,3,2,1,0]
```

List Comprehension

- Syntactic sugar for Higher-Order Functions (w/ select…from…where)
  - to construct lists in a mathematically beautiful way (no more append!)

List Comprehension

- Syntactic sugar for Higher-Order Functions (w/ select…from…where)
  - to construct lists in a mathematically beautiful way (no more append!)

```
age = []
for y in birth_year:
    age.append(2016-y)

senior = []
for y in age:
    if y>60:
        senior.append(y)
```

List Comprehension

- Syntactic sugar for Higher-Order Functions (w/ select…from…where)
  - to construct lists in a mathematically beautiful way (no more append!)

```
age = [2016-y for y in birth_year]
```

```
senior = []
for y in age:
    if y>60:
        senior.append(y)
```

List Comprehension

- Syntactic sugar for Higher-Order Functions (w/ select…from…where)
  - to construct lists in a mathematically beautiful way (no more append!)

```
age = [2016-y for y in birth_year]
```

```
senior = [y for y in age if y>60]
```

List Comprehension

- Syntactic sugar for Higher-Order Functions (w/ select…from…where)
  - to construct lists in a mathematically beautiful way (no more append!)

```
age = map(lambda y: 2016-y, birth_year)
```

```
senior = [y for y in age if y>60]
```

List Comprehension

- Syntactic sugar for Higher-Order Functions (w/ select…from…where)
  - to construct lists in a mathematically beautiful way (no more append!)

```
age = map(lambda y: 2016-y, birth_year)
```

```
senior = filter(lambda y: y>60, age)
```

Why HOF matters?

- Big Data Platforms including Apache Spark (and the principals of Hadoop) are built on functional programming languages
  - has a smaller set of data processing constructs, thus, easier for the platforms to optimize parallelism
  - less prone to data states and copies
  - keep track of provenance (data + transformation)
- As a user, it keeps us more conscious about concurrency and parallelism
  - algorithm design at a high level (mathematically)

Why HOF matters?

- In a nut shell, programming Big Data Platforms ~ data + a series of Higher-Order functions, where we only have control over our functional arguments:
  - reduce(map(filter(map(A,…),…),…),…)
  - A.map().filter().map().reduce()
- Many of built-in HOF maps well to parallel platforms: data parallelism
  - functions are executed in context independent of each other (except reduce, which has a shared memory buffer)
  - multiprocessing module also provides a parallel version of map()

reduce() in Python 3

- Its usage is “discouraged”
  - “it is 99% better to use a for loop instead”
  - It is available in the “functools” module

```
from functools import reduce
```

- But for loop is not always possible for big data platforms
  - Still need an abstract way (specifying “what”, not “how) for aggregation

Questions?

- Big Data Platforms need to abstract data transformation -> Higher-Order functions
  - Think Big Data -> think Functional!
- Resources on HOF:
  - https://wizardforcel.gitbooks.io/sicp-in-python/content/ — 1.6 Higher-Order Functions
  - https://github.com/joelgrus/stupid-itertools-tricks-pydata — Functional Python for Learning Data Science
    - Slides included